Map On Disk

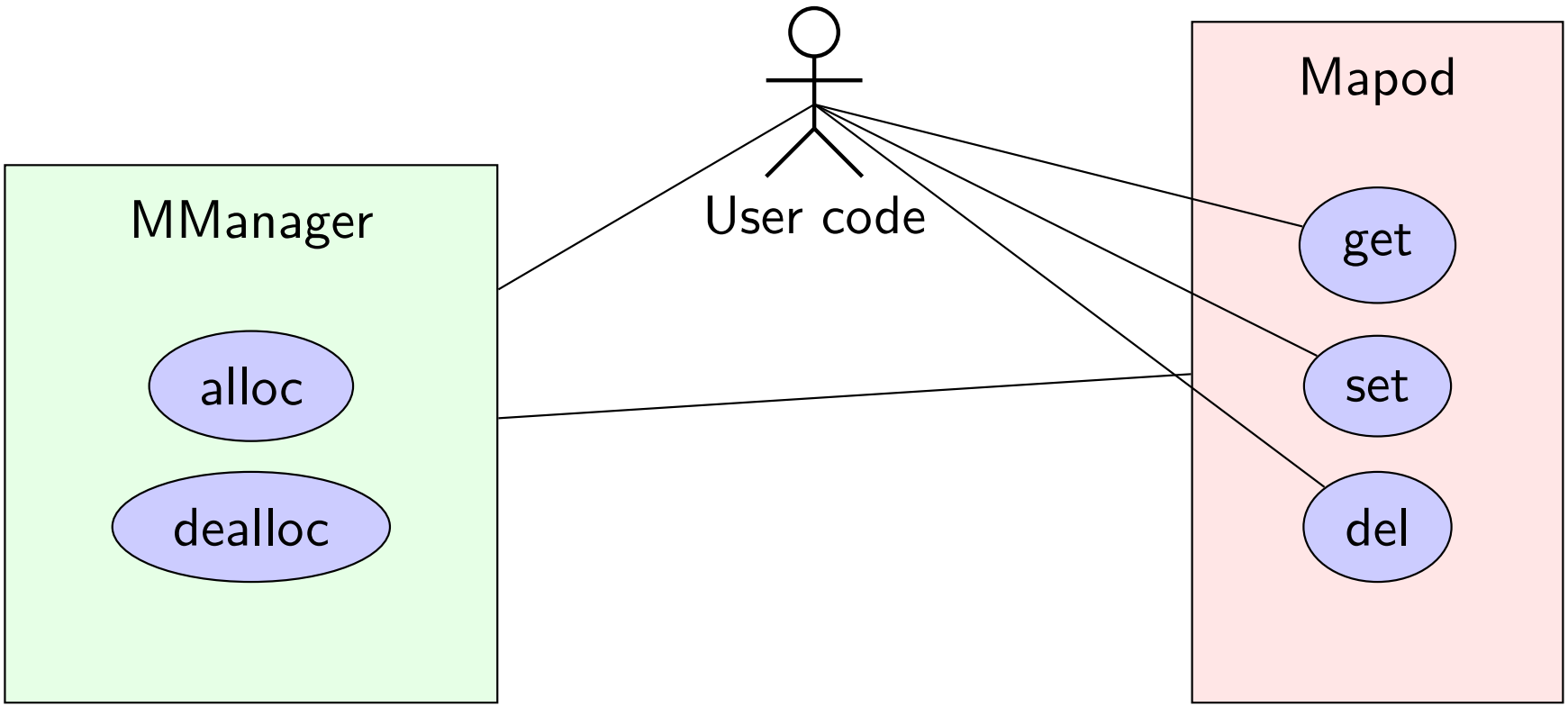
钱泽森

January 2, 2015

Introduction

- Implemented by B+ tree, insert/remove/search are supported
- Templatized, Key and Value can be any type(including user-defined types)
- cache and consistence handled by OS(using `mmap()`)
- Simple serializer included, while user can define their own

Design



Code example

```
#include "mapod.hpp"
#include <iostream>
int main()
{
  //"demomap" is the filename
  Mapod<long, long> map("demomap");
  map.set(123, 321);
  std::cout << "123 is mapped to "
            << map.get(123) << std::endl;
  map.del(123);
}
```

Code example with custom serializer

```
off_t dump(const string &t, MManager &mm)
{
  const off_t off =
    mm.alloc(t.size() + 1);
  char *ptr = (char *)mm[off];
  strcpy(ptr, t.c_str());
  return off;
}

string load(const off_t off, MManager &mm)
{
  const char *ptr = (char *)mm[off];
  return string(ptr);
}
```

```
void destroy(const off_t off, MManager &
{
  mm.dealloc(off);
}

int main()
{
  DLD<string> dld{dump, load, destroy};
  Mapod<string, string> map("map"
  , Mapod<string, string>::compare
  , dld, dld);
  string command;
  string key, t;
  /* remaining code is same as
     the the previous example */
}
```

Correctness Test

1. Use `std::map` as reference
2. Test scenarios where number of entries are 2, 4, 8, 16, ... 1048576
3. For each scenarios, do tests 1, 2, 3, ..., 20 times.
4. In each test, perform n^3 operations, where n is the number of entries. The operation can be insertation or deletion, and is randomly choosed. After each operaion, a random query is performed, and the result is compared to the result given by `std::map`.

Run above test for `<long, long>` and `<string, string>` without error.

Performance Test

Specs

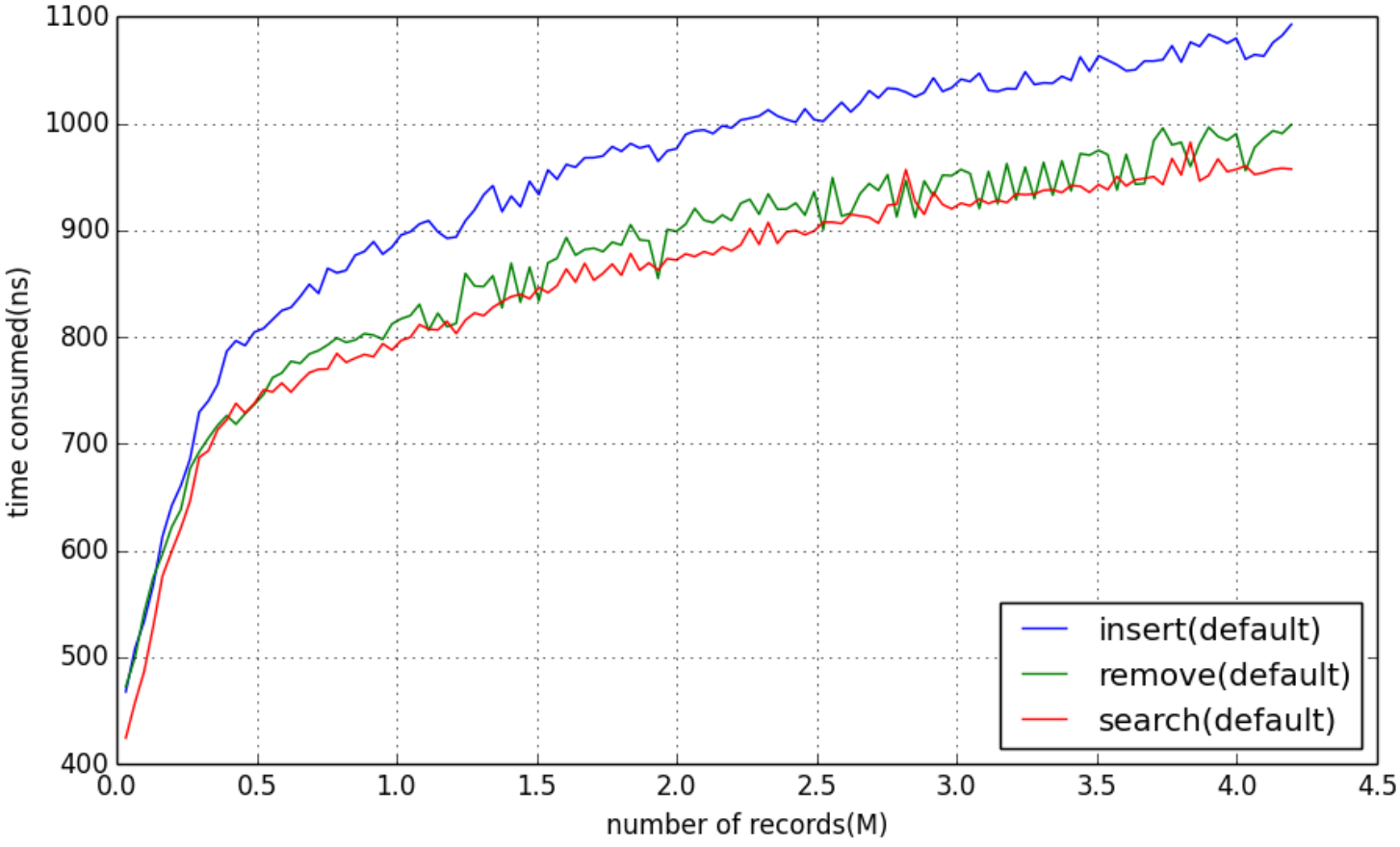
CPU	Intel(R) Core(TM) i3-4010U CPU @ 1.70GHz
CPU cache	3072 KB
Keys	8 bytes each
Values	8 bytes each
Entries	1048576
Raw Size	16 MB (estimated)

Performance Test

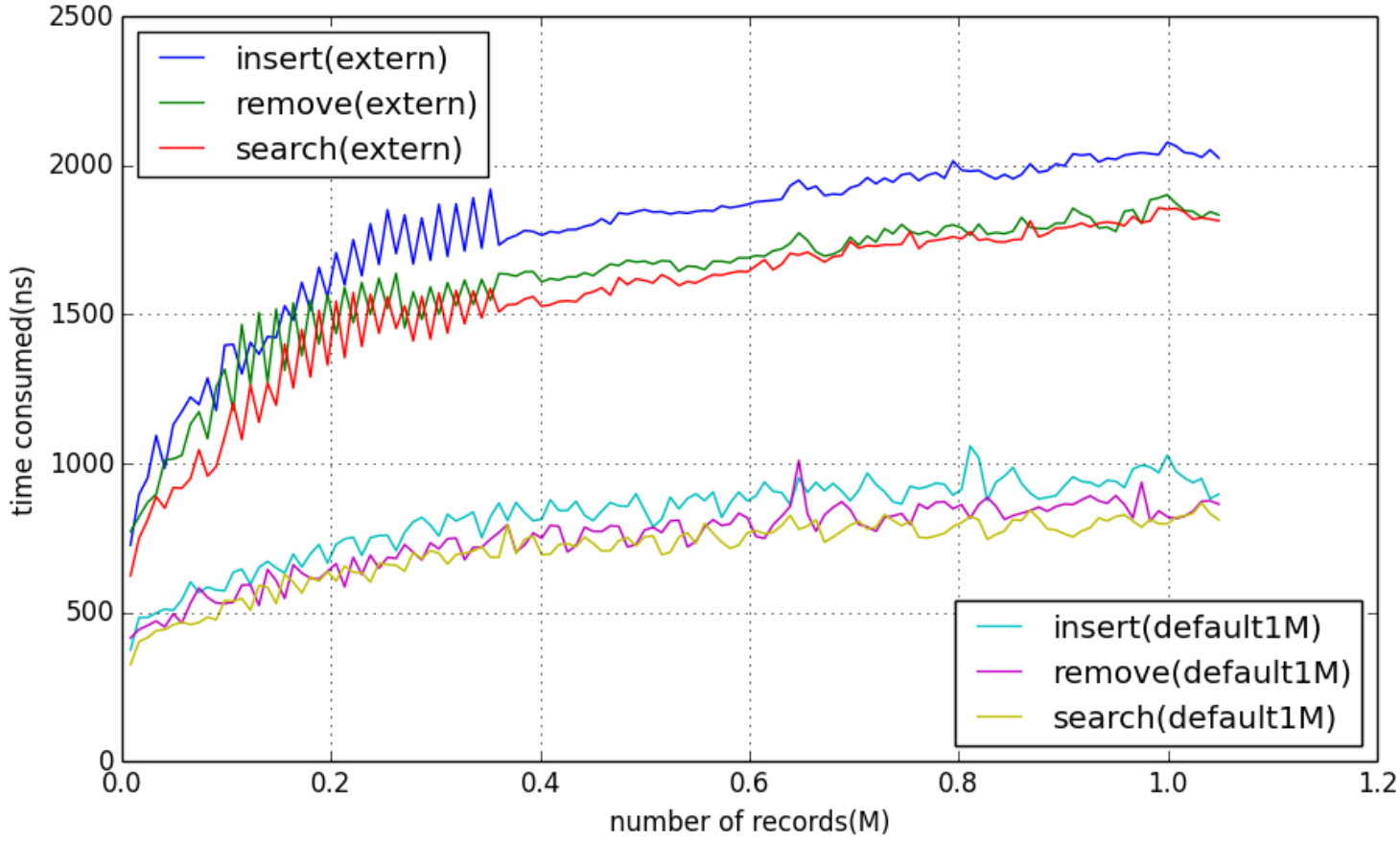
1. Get the number of current entries
2. Insert $2^{13}(8192)$ random entries, and record the time
3. Query $2^{13}(8192)$ random entries, and record the time
4. Insert $2^{13}(8192)$ random entries, and record all the keys inserted
5. Delete entries according to 4, and record the time

Repeat the above steps for $2^{7}(128)$ times, until the number of entries reach $2^{20}(1048576)$. Now we have 128 groups of data indicating the time consumed for a single insert / delete / search, while the number of entries in range $[0, 2^{20}]$.

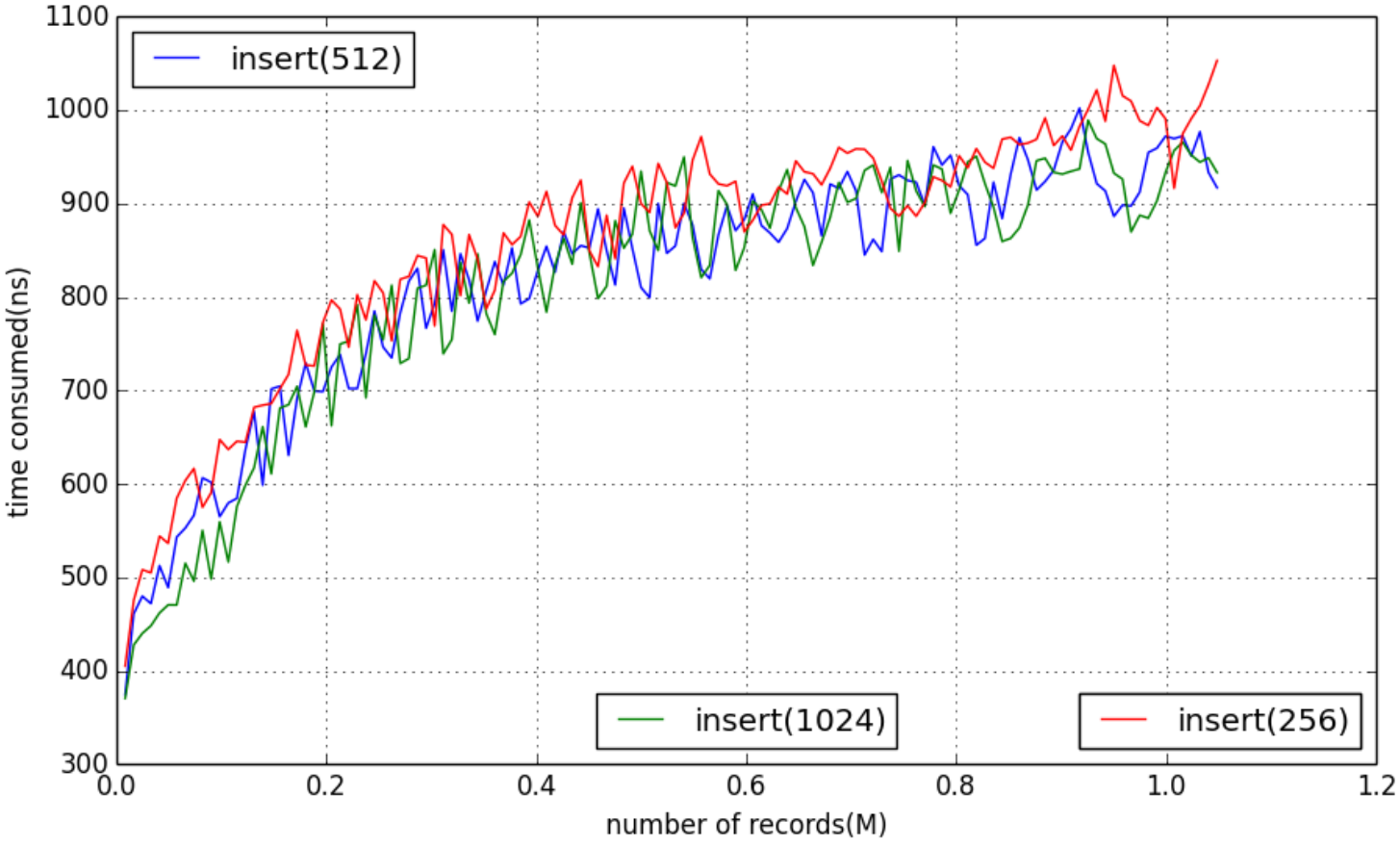
Default Configuration



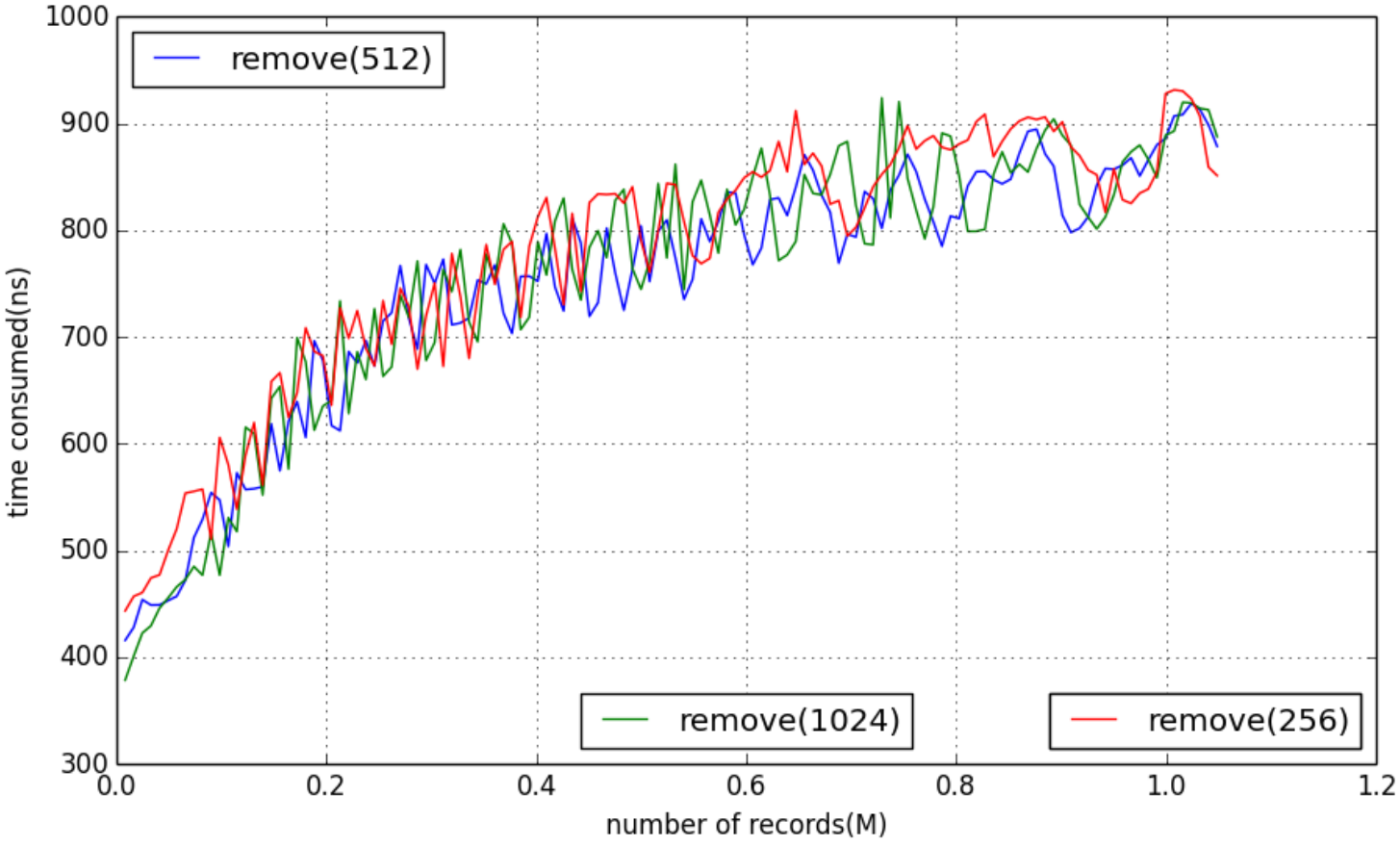
Inline Vs. External



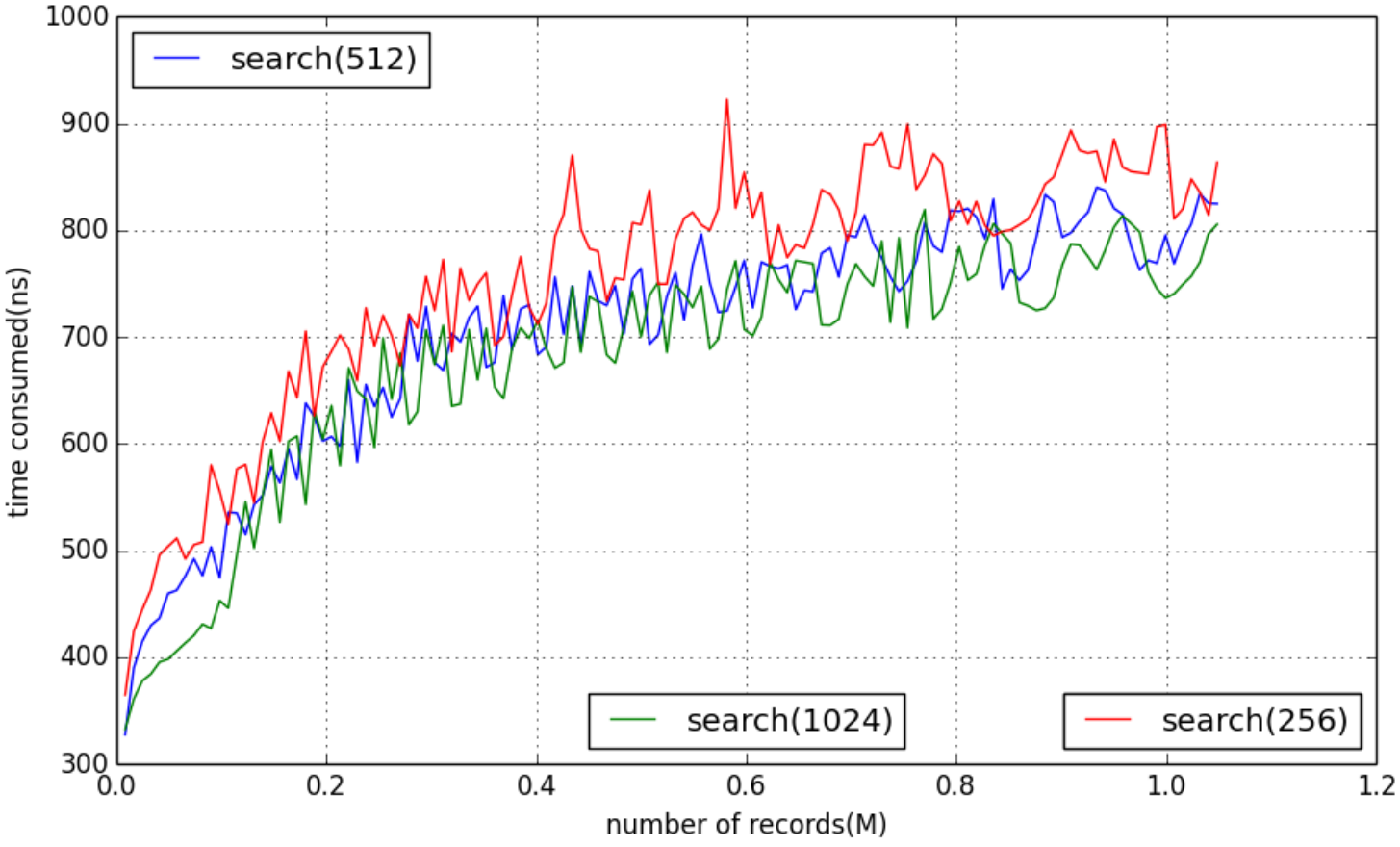
The Optimal `blksize` (insert)



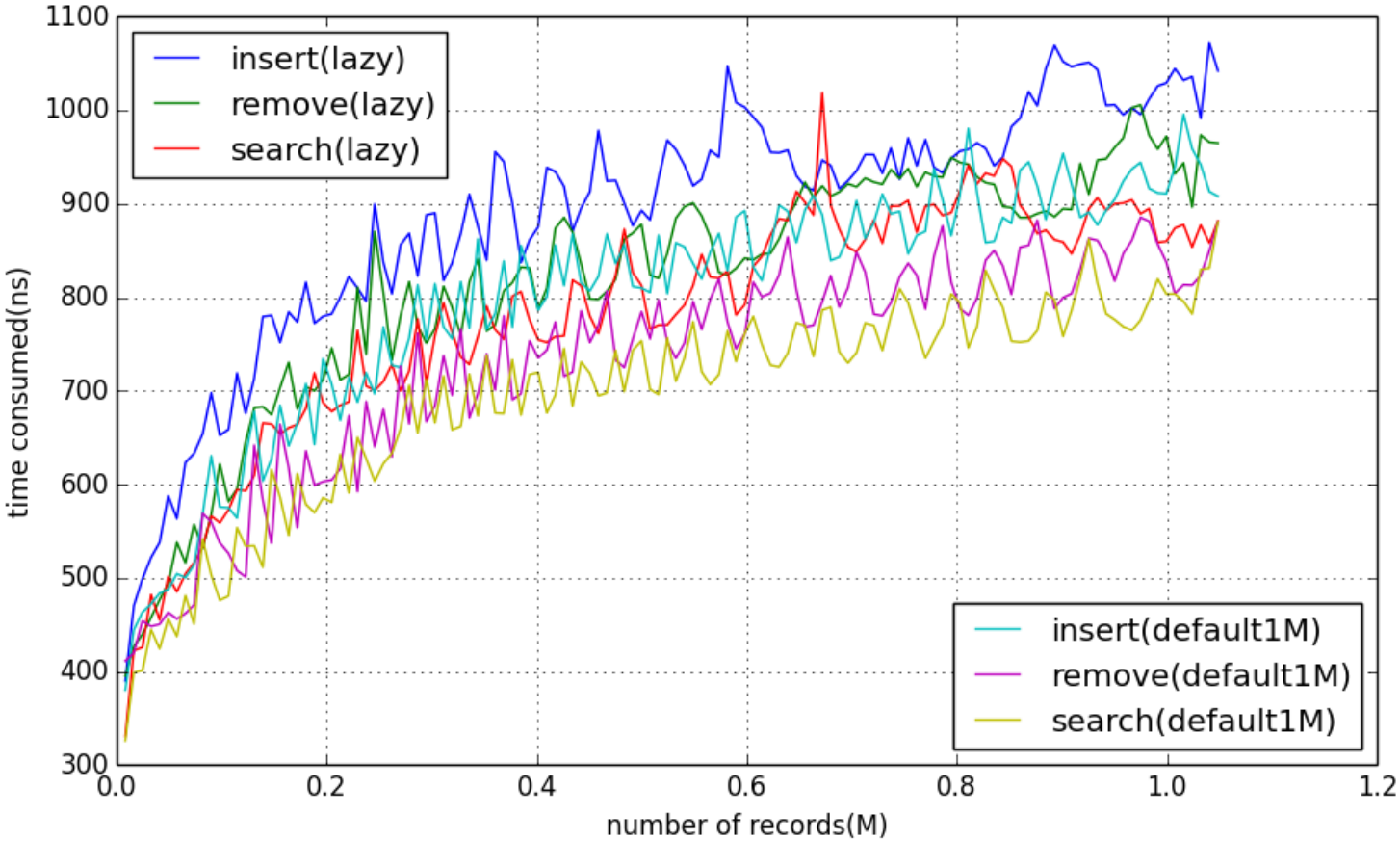
The Optimal `blksize` (remove)



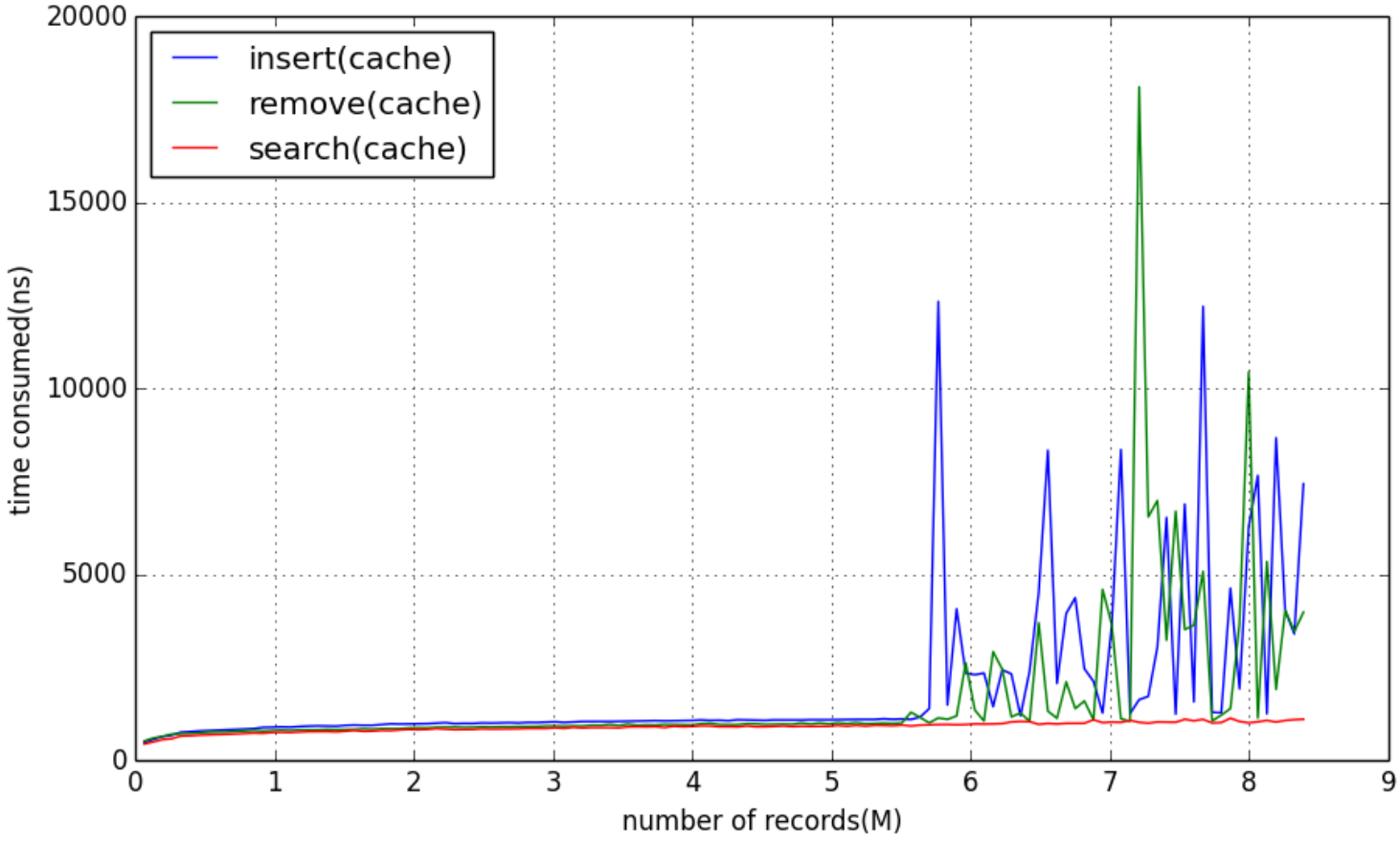
The Optimal `blksize` (search)



Lazy or Not Lazy



OS Cache


insert(cache)
remove(cache)
search(cache)
time consumed(ns)
number of records(M)
0
5000
10000
15000
20000
0
1
2
3
4
5
6
7
8
9

OS Cache

- 5,000,000 of `<long, long>` is actually only about 80MB
- OS cache is for general purpose(not optimized for database)
- Walkaround
  - tune the kernel parameters so it tends to use more cache
  - call `mlock()` on `mmap()`-ed memory, to hint the kernel to preserve the data in the physical memory
    - not work everytime
- Or design our own cache algorithms, which is specialized to database, and should be of high performance
  - too complicated, didn't implement